A Probabilistic based Approach towards Software System Clustering

Anna Corazza\textsuperscript{1}, Sergio Di Martino\textsuperscript{1}, Giuseppe Scanniello\textsuperscript{2}

\textsuperscript{1}Dipartimento di Scienze Fisiche – Sezione Informatica  
University of Naples “Federico II”  
via Cinthia, 80126, Napoli, Italy  
{corazza | dimartino}@na.infn.it

\textsuperscript{2}Dipartimento di Matematica e Informatica  
University of Basilicata  
Viale Dell'Ateneo, 85100, Potenza, Italy  
giuseppe.scanniello@unibas.it

Abstract—In this paper we present a clustering based approach to partition software systems into meaningful subsystems. In particular, the approach uses lexical information extracted from four zones in Java classes, which may provide a different contribution towards software systems partitioning. To automatically weigh these zones, we introduced a probabilistic model, and applied the Expectation-Maximization (EM) algorithm. To group classes according to the considered lexical information, we customized the well-known K-Medoids algorithm. To assess the approach and the implemented supporting system, we have conducted a case study on six open source software systems.

Keywords-component; Probabilistic Model; Software Partitioning; Reverse Engineering; Clustering; Architecture Recovery.

I. INTRODUCTION

The maintenance is usually the largest and the most expensive activity in the software life cycle. Maintenance starts after the delivery of the first version of a software and lasts much longer than the initial development phase [16]. During the maintenance phase, a software system will be continuously changed and enhanced for several reasons, thus decaying its structure [19], [28], [35]. Furthermore, its documentation could become outdated if it is not properly managed. In these cases, issues concerning system structure recovery and documentation are of primary interest within today’s software systems.

In the context of software maintenance, software architecture recovery represents a longstanding and relevant research topic. In fact, a number of approaches, techniques, and tools have been designed and developed to support the architecture recovery and the remodularization of legacy software systems [7], [17], [27], [29]. The majority of them employ clustering algorithms to partition software systems into meaningful subsystems. These approaches generally attempt to discover clusters by analyzing structural dependencies between software artifacts or considering naming information, such as file names [1].

In this paper we present a novel clustering based approach to partition legacy software systems. It exploits only lexical information extracted from the source code to identify meaningful software subsystems. Differently from the previously presented approaches [18], [21], we mine information from four zones in the source code of Java systems, namely (I) comments, (II) Javadoc, (III) class/method identifiers, and (IV) variable identifiers. This information is preprocessed by applying some text processing techniques. As a first formulation of our approach, as subsequent step we clustered the data by a version of the well-known K-Medoids algorithm [23], which has been modified to better handle the specific domain. This solution is based on the assumption that the four previously identified zones within the code can convey information with the same importance. To relax this constraint, we introduced a probabilistic model, and applied the Expectation-Maximization (EM) algorithm [25], to automatically assign weights to the four zones, before applying the clustering algorithm. The EM technique, in fact, is an iterative algorithm aiming at finding estimates for the model parameters corresponding to a (local) maximum likelihood of the training data.

The approach has been implemented in a prototype of a supporting software system, intended as an Eclipse plug-in. The proposal has been evaluated on six open source software systems implemented in Java. We also carried out a preliminary evaluation of the approach, comparing them with the result obtained without applying the probabilistic model, and with those presented by in a comparative of clustering algorithms provided by Bittencourt and Guerrero in [3]. The evaluation revealed that the approach and the tool prototype enable the software engineer to obtain good results according to the considered criteria.

The remainder of the paper is organized as follows: related work is discussed in Section 2. In Section 3 we
describe the approach while Section 4 presents the case study and discusses the obtained results. Finally, conclusions and possible future direction for our research conclude the paper.

II. RELATED WORK

The maintainability of a software system increases in case its software architecture is well documented. Unfortunately, this kind of documentation may become outdated when software engineers do not consistently maintain it with the changes made [4], [10]. In this context, reverse engineering tools may be adopted to retrieve and align it with the implementation of the software system.

The majority of the reverse engineering tools proposed in the literature are based on clustering algorithms [2], [5], [18], [24], [32]. For example, Wiggerts [32] presents clustering algorithms commonly used in the past to group entities into software subsystems. Indeed, he provides a theoretical background for the application of cluster analysis in systems remodularization. To this end, three concerns are mainly addressed: the entities to be analyzed, similarity measures to compare the entities, and the clustering algorithm to apply. Anquetil and Lethbridge [2] extend the work by Wiggerts [32] presenting a comparative study of different hierarchical clustering algorithms and analyze their properties with regard to software remodularization.

Maqbool and Babri in [24] highlight hierarchical clustering research in the context of software architecture recovery and remodularization. They pose special emphasis on the analysis of various similarity and distance measures that could be used in the software clustering in general and in the software remodularization in particular. The main contribution of the paper is, however, the analysis of two clustering based approaches and their experimental assessment on some large software systems.

In [26] a clustering system, named Bunch, has been proposed and analyzed. Differently from us, to produce a decomposition of a system in subsystems Bunch uses search techniques to partition the graph representing software entities and their relations. Indeed, the tool is based on several heuristics to navigate through the search space of all possible graph partitions. To evaluate the quality of graph partitions and to find a satisfactory solution, the tool uses fitness functions. The tool effectiveness has been assessed using qualitative and quantitative empirical investigations. Even, in [8] a structural approach based on genetic algorithms is proposed to group software entities in clusters. The effectiveness of the approach has been assessed on a small software system. Mahdavi et al. in [20] propose a search-based approach to the automated module clustering problem. Indeed, the approach uses dependencies between modules to maximize cohesion within each cluster and to minimize coupling between clusters.

To quantitatively assess the decompositions produced by clustering algorithms, Tzerpos and Holt in [30] propose the MoJo distance measure, which is defined as the minimum number of move and joint operations to turn a source cluster into a target one. Indeed, the computation of the MoJo measure is based on a heuristic that approximates the exact measure values. Successively, Wen and Tzerpos in [33] propose an optimal algorithm to calculate the exact MoJo distance in polynomial time. The defined measure has been successively used to assess the result of several clustering-based approaches. For example, Tzerpos and Holt use the MoJo distance in [31] to study the stability and the quality of a number of software clustering algorithms. The comparison among clustering algorithms is conducted generating randomly “perturbed” versions of an example system. Differences between the partition identified by the clustering algorithms and the original partition of the system are measured.

Wu et al. in [34] present a comparative study of a number of clustering algorithms: (i) an agglomerative clustering algorithm (based on the Jaccard coefficient and the complete linkage update rule) using 0.75 and 0.90 as cutting points; (ii) an agglomerative clustering algorithm (based on the Jaccard coefficient and the single linkage update rule) using 0.75 and 0.90 as cutting points; (iii) an algorithm based on program comprehension patterns that tries to recover subsystems that are commonly found in manually-created decompositions of large software systems; and (iv) a customized configuration of an algorithm implemented in Bunch [22]. The authors compare these algorithms on five large C/C++ open source systems. Similarly to us, Authoritativeness and non-extremity of cluster distribution have been considered to assess the achieved results. The Authoritativeness is assessed using the MoJo distance, while the non-extremity of cluster distribution has been evaluated using NED (Non-Extreme Distribution). Similarly, in [3] an empirical study is presented to evaluate four widely known clustering algorithms on 15 systems implemented in Java and C/C++.

Generally, reverse engineering approaches are focused on structural information to recover software architectures. Nevertheless, the domain knowledge of the developers is generally embedded in the code comments. For such a reason, Kuhn et al. in [18] describe an approach to group software artifacts based on Latent Semantic Indexing (LSI) [6], [13]. The approach is language independent and tries to group source code containing similar terms in the comments, similarly to our approach. The approach is implemented within a tool named Hapax, which is built on top of the Moose reengineering environment [9]. Case studies are used to assess the approach and the tool support. The achieved results have not been quantitative evaluated. Our approach is different with respect to the ones presented above as we mine information extracted from four different zones in source existing source code. Since these zones may provide different contributions, we automatically weighted them using a probabilistic model and the EM algorithm.

Maletic and Marcus [21] propose an approach based on the combination of two dimensions (e.g., semantic and structural) to support the comprehension tasks within the maintenance and reengineering of software systems. From the semantic point of view they consider problem and development domains. On the other hand, the structural dimension refers to the actual syntactic structure of the program along with the control and dataflow that it represents. Software entities are semantically compared
using LSI, while file organization is used to get structural information. To assess the effectiveness of the approach some case studies on a version of Mosaic are presented and discussed. Adritos and Tzerpos in [1] present LIMBO, a hierarchical algorithm for software clustering. The clustering algorithm considers both structural and non structural attributes to reduce the complexity of a software system by decomposing it into clusters. The authors also apply LIMBO to three large software systems.

III. THE APPROACH

The approach we are proposing towards software system partitioning relies on the lexical information contained within the source code. We started from the observation that nowadays more and more software projects are being developed using naming conventions. As a consequence more expressive names for classes, methods and variables, as well as for text in comments are used. This is also true with agile methods, such as eXtreme Programming, where the use of meaningful naming in the code is one of the pillars for the development.

Our idea is to take advantage of the precious, but often unexploited, lexical information given by the programmers for maintenance activities, to identify related artifacts within a software system in an automatic fashion. To infer these relationships, we exploit in a new way some of the state-of-the-art techniques for text handling, in order to extract as much information as possible from the code. In particular, we consider every zone where a programmer can introduce meaningful information in the source code, namely:

1. Natural language comments
2. Javadoc comments
3. Method and Class signatures
4. Variable identifiers

To identify meaningful subsystems in a software system, our approach encompasses three main steps:

1. Feature Extraction. This is the first step, where terms are extracted from the code, they are preprocessed, and their relevance is computed using the tf-idf indicator [23].
2. Weighting with a Probabilistic Model. Assign a specific weight to each of the four zones, using the Expectation – Maximization algorithm.
3. Clustering. This is the last step, where “related” software artifacts are clustered together thanks to a modified version of the K-Medoids algorithm.

These steps are detailed in the three next sections, while in Section 4 we present the evaluation of the approach.

A. Feature Extraction

The first task is to extract lexical information from the source code and organize it in some meaningful structure, suitable for further processing. To this aim, some preprocessing is needed before arranging the mined information.

First of all, we had to decide at which granularity the input files are segmented. Since we are dealing with remodularization tasks, we choose the class as atomic element, so that each class in our software repository is considered as a different item (which we call document by adopting the Information Retrieval terminology) [23]. Thus, we read the whole source code of the system under investigation, defining a new document for each class. Then we started by extracting only the information that was relevant for our goal.

To explain how we designed the extraction task, let us consider the following fragment of code:

```java
1/** 2 * The responsibility of this class is to 3 * manage the airplane tickets reservation. 4 */ 5 public class TicketManagement 6 { 7  boolean ticketReservationStatus = false;
```

As first step of the preprocessing phase, all the language keywords are removed. In our case, we eliminate terms like `public`, `class`, `boolean`, `this`, `int`, etc...

As second step, we identify all the significant terms in the code. To this aim, it is worth noting that, while in the case of comments the same preprocessing which is usually applied to textual documents is sufficient, when dealing with software code, some additional considerations are needed: for example, the variable `ticketReservationStatus` is composed by three words, which should be properly identified and separately handled [15]. Thus, identifiers have to be split in their lexical segments, basing on the naming conventions adopted by the programmers. In our approach, we can handle capitalized letters (Camel/Pascal case) and underscores as separators. In our example, at the end of this task, we have identified terms like `ticket`, `management`, `the`, `responsibility`, `set`, etc...

In the next step, all the most common terms in the natural language (stopwords) are removed [25], since they do not contain important significance to be used later. For example, terms like `the`, `is`, `to`, `etc`… are removed.

Finally, all lexical items are normalized by applying the Porter stemmer [23], resulting in the terms to be included into the dictionary. For instance, both the words `management`
and manage lead to the common radix manage, that will be added into the dictionary.

To record also the information about the zone of the code containing each a word, a suffix (_01, _02, _03 or _04, respectively for Javadocs, comments, signatures and variables) is appended to each term before it is added to the dictionary. Hence, it is worth noting that in our approach, a word appearing in two zones in handled as two different terms.

The final table containing the number of term occurrences for the given example is:

<table>
<thead>
<tr>
<th>Token</th>
<th>tf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Airplane_01</td>
<td>1</td>
</tr>
<tr>
<td>Airplane_02</td>
<td>2</td>
</tr>
<tr>
<td>Constructor_02</td>
<td>1</td>
</tr>
<tr>
<td>Manage_01</td>
<td>1</td>
</tr>
<tr>
<td>Manage_03</td>
<td>2</td>
</tr>
<tr>
<td>Reservation_01</td>
<td>1</td>
</tr>
<tr>
<td>Reservation_02</td>
<td>2</td>
</tr>
<tr>
<td>Reservation_03</td>
<td>2</td>
</tr>
<tr>
<td>Reservation_04</td>
<td>1</td>
</tr>
<tr>
<td>Responsibility_01</td>
<td>1</td>
</tr>
<tr>
<td>Status_02</td>
<td>2</td>
</tr>
<tr>
<td>Status_04</td>
<td>1</td>
</tr>
<tr>
<td>Ticket_01</td>
<td>1</td>
</tr>
<tr>
<td>Ticket_02</td>
<td>2</td>
</tr>
<tr>
<td>Ticket_03</td>
<td>2</td>
</tr>
<tr>
<td>Ticket_04</td>
<td>1</td>
</tr>
<tr>
<td>Value_04</td>
<td>1</td>
</tr>
</tbody>
</table>

Of course, in this toy example frequencies are much lower than in a more realistic case. The idf, on the other hand, depend on the whole document collection, which, in this case, is the whole software system, and therefore cannot be computed on the basis of a single class.

After this segmentation, a vector of real numbers is built to represent each document, so that a point is associated to each document, according to the classical Vector Space Model (VSM) [23]. In this way, subsequent clustering algorithms can be based on geometrical principles, where similarity measures can be based on point distances. Each element of the vector, corresponding to a term in the dictionary extracted from the source code, is represented by a positive or null real number and gives an idea of the “importance” of the term with respect to the document. Another way of looking at the problem instantiation is by collecting all vectors in a matrix, where columns correspond to documents and rows to terms (the so-called term-document matrix).

The document model assumed by this representation is called bag-of-words [23], since each document is represented as a multi-set of words (that is a set of words counted with the number of their occurrences), disregarding all information about their order or syntactic structure. For the weight associated to each pair <term, document> we considered the most widely adopted one in information retrieval, namely the term frequency–inverse document frequency, also known as tf-idf, which, for every term \( t \) and document \( d \) is defined by:

\[
\text{tfidf}(t_i, d_j) = \frac{\text{tf}(t_i, d_j)}{\text{df}(t_i)} \log \frac{|C|}{\text{df}(t_i)}
\]

where \( \text{tf}(t_i, d_j) \) is the number of occurrences of the term \( i \) in the document \( j \), \(|C|\) is the total number of documents and \( \text{df}(t_i) \) is the number of documents which contain the term \( t \). The tf-idf is null only when the term does not appear in the document or when the term appears in all documents. In both cases, the presence of the term in the document is semantically irrelevant. In all other cases, it is greater than zero because \( \text{df}(t_i) \leq |C| \). Higher weights are assigned either to terms with a high number of occurrences in the document or which appear in a small number of documents in the collection. The dictionary is the list of all normalized different words extracted from the document.

Finally, it is worth noting that in this artifact example, the word “airplane” does not appear within the code, but only in the comments. Thus, approaches discarding comments will lose this key information.

B. Weighting with a Probabilistic Model.

The bag-of-words approach above described makes the assumption that a word in the source code has the same “importance” no matter where it is placed. Indeed, if a clustering algorithm deals only with the tf-idf score of a word, it cannot take into account the zone where that term has been extracted. In other words, a term used in the name of a class is considered as important as the same term used within a comment, but in our programming experience, this assumption doesn’t hold always: usually we programmers take much more care in placing a word in a class name rather than placing the same word in a comment.

Starting from this consideration, we envisioned an approach to assign different importance to the four considered zones of code. On the other hand, since we did not want to bias the result, relying only on our experience, we introduced a probabilistic model, and then the Expectation-Maximization algorithm, to automatically compute the weights for each of the zones for the specific source code under consideration. In this way, the approach can automatically identify a suitable weighting for each zone of code in the software system to study.

More formally, for each term in the dictionary we linearly combined the weight corresponding to each different zone. To identify the best values to be assigned to the parameters of this linear combination, we used the maximum likelihood criterion, and assigned to the weights the values which maximize the probability of the training data.

If we look at the \( N \) zones (four in our case) as a partition of documents, the probability of a document \( d_i \) is given by the product of probability of all zones, and can be written as follows.

\[
p(d_i) = \prod_{k=1}^{N} p(d_i, z_k) = \prod_{k=1}^{N} p(z_k)p(d_i|z_k)
\]

In such model, the \( p(z_k) \) is the weight of the \( k^{th} \) zone in the documents, while \( p(d_i|z_k) \) is the probability of the document conditioned on the zone. In conjunction with the bag-of-words model, this can be decomposed in the product

\[
S_i = \sum_j \text{tfidf}(t_j, d_i) \cdot p(z_j)
\]
of probabilities of each word given the zone. However, we do not explode the formulas for the sake of clarity. The interested reader can find further information in [23] and [25].

We assume that this term has a Gaussian distribution with average $\mu_k$ and variance $\sigma_k$: $p(d_i | z) = G(d_i ; \mu_k, \sigma_k)$ so that equation above becomes a Gaussian mixture:

$$p(d_i) = \sum_{k=1}^{N} \pi_k G(d_i | \mu_k, \sigma_k)$$

To identify the model, we need to estimate the set of parameters $(\pi_k, \mu_k, \sigma_k)$, all of them of size $N$, on the basis of the document collection, where we can compute frequencies. As said above, we opted for the maximum likelihood estimation, probably the widest used estimation criterion. However, since we cannot observe the frequencies corresponding to each particular zone, but only cumulative, the estimation is not direct, but need an iterative procedure, namely the Estimation-Maximization algorithm [25], a well-known algorithm with good convergence characteristics. The resulting $\pi_k$ are then used to weigh the contribution of each zone to the term relevance, multiplying the tf-idf parameter for these weights.

C. The Clustering algorithm

Before applying the clustering algorithm, a document-document matrix is computed, whose elements give the similarity between each pair of classes of the system to study. Since we adopted a cosine-based similarity measure, such operation can be easily performed by multiplying the transposed term-document matrix with the matrix itself.

To group classes (or entities) into clusters, we developed a variation of the K-Medoids algorithm. This is a clustering algorithm similar to the classical K-Means [14], except that each cluster is built around a really existing point in the set instead than the mean of the cluster elements, which could correspond with no actual point. The building of clusters around an actual item instead of the mean makes the algorithm more robust with respect to outliers. Moreover, since the eventual clustering strongly depends on the initial choice of medoids, which is random, to avoid that such randomness leads to unbalanced solutions, we introduced a novel halting criterion. In this way, we avoid the risk of resulting in extremely small or extremely large clusters, which makes sense in the context of software modules, where it is uncommon to have packages with a reduced number of classes. Indeed, the original K-Medoids algorithm starts with a random choice of the k medoids and iterates reassigning at each step all points to the closest medoids, and then recomputing the medoids. Unlucky initial configurations could result in a partition including too small clusters: in the variant of the algorithm proposed here and presented in Figure 1. The whole procedure is repeated until a final solution with no extreme clusters or a maximum number of iterations (numIter) is attained. Even when the procedure is halted because the maximum number of iterations has been attained, the best solution among all the ones found is output.

In this way, the probability of having repeated unlucky initial configuration becomes extremely small. As in other similar works, $K$ (the number of clusters) is set to 10% of the number of classes. The rationale for choosing this value relies on the fact that in this way the clustering-based approach can be automated. However, future work will be devoted to define a possible strategy to select a suitable value for $K$, depending on the specific system to study.

```
if (newTotalDistances == totalDistances) then
  if compute.size.all.clusters(C, size) then
    return(k, M, C)
  else if iter > numIter then
    if newTotalDistances > bestDistances then
      load.context()
    end if
    return(k, M, C)
  else
    if newTotalDistances < bestDistances then
      save.context(k, M, C)
      bestDistances ← newTotalDistances
    end if
    M ← choose.random.medoids(D)
  end if
end if
```

Figure 1. Pseudocode of the modified K-Medoids

IV. Case Study

This section describes the design underlying the case study to assess the proposed approach. The achieved results are presented and discussed as well.

A. Design

The assessment of the overall quality of the clustering based process is generally a critical issue. In this paper, we consider two criteria that are summarized as follows:

- **Authoritiveness** – it regards the resemblance between the software clusters identified by the tool and an authoritative partition (i.e., the decomposition performed by the original developers). The clusters produced by the approach should resemble as much as possible the groups of entities within the authoritative partition.

- **Non-extremity cluster distribution** (NED) – it aims at investigating whether the approach identifies a partition whose clusters have a size distribution exhibiting no extreme values. A good approach should avoid clusters with too many or too few software entities.

According to these two criteria, we formulate the two following research questions:

Q1: Does our baseline system\(^1\) outperform other clustering-based remodularization strategies?

---

\(^1\) We defined as “baseline system” the approach using the K-Medoids clustering algorithm on the lexical features, without zones and weights.
Q2: Does the introduction of the zones weighted by the probabilistic model improve the baseline system?

1) The Used Measures

To evaluate the Authoritativeness we had to measure the similarity between the partitions proposed by our approach and those in the authoritative partitioning. The MoJo measure has been defined to this aim. In particular, let $A$ be the automatically identified partition and $B$ the authoritative partition, the monodirectional $\text{MoJo}(A,B)$ is computed as the minimum number of join and move operations to turn $A$ into $B$ (see [33] for details). In order to make results comparable among software systems with different sizes, we used a normalized version of MoJo, namely the $\text{MoJoSim}[3]$

$$\text{MoJoSim}(A,B) = 1 - \frac{\text{MoJo}(A,B)}{n}$$

where $n$ is the number of entities to be clustered. We have used here the polynomial version of $\text{MoJoSim}$, available at http://www.cse.yorku.ca/~bil/downloads.

An authoritative partition is often hard to find. This is because either the documentation is lacking or it does not accurately reflect the software architecture. Accordingly, we compute the authoritative partition performing the following three steps:

1. create the subsystem hierarchy based on the directory structure (each directory represents a subsystem);
2. merge a subsystem with its parent subsystem in case it contains a number of source files less than or equal five;
3. create a cluster with each subsystem in the subsystem hierarchy.

It is worth mentioning that the lower is the MoJo distance, the more the identified clusters resemble groups of entities within the authoritative partition.

The non-extremity of cluster distribution has been assessed through the NED measure, which is defined as follows:

$$\text{NED} = \sum_{i=1, i \text{ not extreme}}^{k} n_i / n$$

where $k$ is the number of clusters identified by the tool, while $n_i$ represents the size of the $i^{th}$ identified cluster. Finally, $n$ is the number of classes of the analyzed software system. Similarly to [3], clusters with less than 5 or more than 20 entities are considered as extreme lower and upper limits, respectively. It is worth noting that the larger the NED value is, the more non-extreme is the size distribution of the clusters.

2) The Dataset

To conduct our preliminary investigation on the effectiveness of the proposal, we have used six open source Java software systems that are a superset of those analyzed in [3]. In the following, we briefly describe these systems:

- **EasyMock** is a tool for Test-Driven Development. It provides Mock Objects for interfaces by generating them on the fly using Java’s proxy mechanism.
- **JabRef** is an open source bibliography reference manager.
- **JavaGroups** is a group toolkit for reliable multicast communication.
- **JEdit** is a text editor, suited to support programming tasks.
- **JUnit** is a unit testing framework for the Java programming language.
- **JUnit** is a vocabulary learning tool.

TABLE II provides some descriptive statistics of the considered software systems. The first and second columns report the names and the versions of the analyzed software systems, while the third one shows the number of classes for each one. The last two columns report the number of thousand of lines of code (KLOC), and thousands of lines of comments (CKLOC).

<table>
<thead>
<tr>
<th>Systems</th>
<th>Version</th>
<th>Classes #</th>
<th>KLOC</th>
<th>CKLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easymock</td>
<td>2.4</td>
<td>65</td>
<td>3.06</td>
<td>0.23</td>
</tr>
<tr>
<td>JabRef</td>
<td>2.4 β2</td>
<td>550</td>
<td>74.74</td>
<td>16.49</td>
</tr>
<tr>
<td>JavaGroups</td>
<td>2.6.3 GA</td>
<td>511</td>
<td>83.07</td>
<td>13.06</td>
</tr>
<tr>
<td>JEdit</td>
<td>4.2</td>
<td>498</td>
<td>87.68</td>
<td>22.55</td>
</tr>
<tr>
<td>JUnit</td>
<td>4.5</td>
<td>27</td>
<td>2.74</td>
<td>1.3</td>
</tr>
<tr>
<td>Jvlt</td>
<td>1.1.1</td>
<td>288</td>
<td>18.82</td>
<td>0.47</td>
</tr>
</tbody>
</table>

The dataset is quite heterogeneous in terms of provided features and the size of these applications, in terms of number of classes, can highly vary, ranging from a minimum of 27 classes for JUnit, till 550 for JabRef. The same thing is for the lines of code, ranging from 2,300 to 140,700.

V. RESULTS

The step of features-extraction produces the normalized terms distributed in the four zones, as described in TABLE III.

<table>
<thead>
<tr>
<th>System</th>
<th>javadoc</th>
<th>Comments</th>
<th>Signatures</th>
<th>Identifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>EasyMock</td>
<td>16.3%</td>
<td>21.26%</td>
<td>27.1%</td>
<td>85.3%</td>
</tr>
<tr>
<td></td>
<td>0.9</td>
<td>0.005</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>JabRef</td>
<td>17.1%</td>
<td>31.1%</td>
<td>16.2%</td>
<td>38.5%</td>
</tr>
<tr>
<td></td>
<td>0.64</td>
<td>0.20</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>JavaGroups</td>
<td>27.6%</td>
<td>5.7%</td>
<td>23.4%</td>
<td>43.2%</td>
</tr>
<tr>
<td></td>
<td>0.48</td>
<td>0.25</td>
<td>0.10</td>
<td>0.17</td>
</tr>
<tr>
<td>JEdit</td>
<td>28%</td>
<td>30.7%</td>
<td>12.8%</td>
<td>28.6%</td>
</tr>
<tr>
<td></td>
<td>0.32</td>
<td>0.10</td>
<td>0.24</td>
<td>0.34</td>
</tr>
<tr>
<td>JUnit</td>
<td>59.1%</td>
<td>1.6%</td>
<td>1%</td>
<td>23.3%</td>
</tr>
<tr>
<td></td>
<td>0.008</td>
<td>0.67</td>
<td>0.15</td>
<td>0.17</td>
</tr>
<tr>
<td>Jvlt</td>
<td>9.3%</td>
<td>4.5%</td>
<td>9.9%</td>
<td>0.3%</td>
</tr>
<tr>
<td></td>
<td>0.02</td>
<td>0.55</td>
<td>0.18</td>
<td>0.24</td>
</tr>
</tbody>
</table>

In particular, in this table we report the percentage of terms extracted by each zone (given 100 the total number of terms extracted for each software system), and the associated weight $\pi$ computed by the EM algorithm. As we can see, there is a high variance in these percentages. For example,
the percentage of terms extracted from comments can range from 1.2% in JUnit, till 30.6% in JEdit. The same happens for JavaDocs, where this percentage ranges from 2.3% (Jvlt) to 59.1% (JUnit).

A. Q1: Effectiveness of the Baseline System

As a first assessment on the effectiveness of the approach, we report the results of the baseline system, i.e. lexical features extracted by the code, and clustered together with the K-Medoids. The results are reported on TABLE IV.

TABLE IV. THE RESULTS FOR THE BASELINE SYSTEM

<table>
<thead>
<tr>
<th>Systems</th>
<th>Authoritativeness</th>
<th>NED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easymock</td>
<td>0.74</td>
<td>0.95</td>
</tr>
<tr>
<td>Jabref</td>
<td>0.5</td>
<td>0.98</td>
</tr>
<tr>
<td>Javagroups</td>
<td>0.5</td>
<td>0.95</td>
</tr>
<tr>
<td>JEdit</td>
<td>0.47</td>
<td>0.97</td>
</tr>
<tr>
<td>JUnit</td>
<td>0.53</td>
<td>1.00</td>
</tr>
<tr>
<td>Jvlt</td>
<td>0.83</td>
<td>0.95</td>
</tr>
</tbody>
</table>

As benchmark to assess the effectiveness of the baseline system, we looked at the work presented by Bittencourt and Guerrero, in [3]. Indeed, the authors assessed the performances of four clustering algorithms in terms of NED and Authoritativeness, on four Java systems, basing on syntactic features extracted from the bytecode.

These compared algorithms were:
- Edge betweenness clustering (eb) [12];
- k-means clustering (km) [14];
- Modularization quality clustering (mq) [22];
- Design structure matrix clustering (dsm) [11].

Since the applications considered by these authors are a subset of the ones we considered, we can compare our results with those they presented in [3]. However, note that we are not able to conduct a punctual comparison because authors specified neither the analyzed versions nor the corresponding Authoritativeness values (only intervals are shown).

In TABLE V there are the results of this comparison, in terms of ranges of values obtained by the various algorithms for NED and Authoritativeness.

TABLE V. COMPARISON WITH BITTENCOURT AND GUERRERO

<table>
<thead>
<tr>
<th>Bittern-court and Guerrero</th>
<th>NED</th>
<th>Authoritativeness</th>
</tr>
</thead>
<tbody>
<tr>
<td>eb</td>
<td>near zero</td>
<td>0.20 - 0.60</td>
</tr>
<tr>
<td>km</td>
<td>0.30 - 1.00</td>
<td>0.40 - 0.80</td>
</tr>
<tr>
<td>mq</td>
<td>0.10 - 0.50</td>
<td>0.30 - 0.70</td>
</tr>
<tr>
<td>dsm</td>
<td>0.10 - 0.70</td>
<td>0.35 - 0.75</td>
</tr>
<tr>
<td>Our approach</td>
<td>K-Medoids</td>
<td>0.95 - 1.00</td>
</tr>
<tr>
<td></td>
<td>K-Medoids + EM</td>
<td>0.96 - 1.00</td>
</tr>
</tbody>
</table>

Another observation is that the baseline system provides clusters whose Authoritativeness is better than any other clustering algorithm considered in [3] and applied on syntactic features.

B. Q2: Improvement obtained by the weighted zones

The results gathered by introducing the zones, weighted with the probabilistic model, are reported in TABLE VI.

TABLE VI. THE RESULTS FOR Q2

<table>
<thead>
<tr>
<th>Systems</th>
<th>Authoritativeness</th>
<th>NED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easymock</td>
<td>0.75</td>
<td>0.97</td>
</tr>
<tr>
<td>Jabref</td>
<td>0.52</td>
<td>0.98</td>
</tr>
<tr>
<td>Javagroups</td>
<td>0.55</td>
<td>0.95</td>
</tr>
<tr>
<td>JEdit</td>
<td>0.5</td>
<td>0.97</td>
</tr>
<tr>
<td>JUnit</td>
<td>0.62</td>
<td>1.00</td>
</tr>
<tr>
<td>Jvlt</td>
<td>0.83</td>
<td>0.97</td>
</tr>
</tbody>
</table>

From these values, we can notice that the application of the zones and the EM algorithm always provides results that are better or equal than without its application. In particular, its application improves the results up to 17%. Thus, even if the probabilistic model does not appreciably improve the best values, we have a mean enhancement of 6%, with a peak of 17% on JUnit.

Figures 2 and 3 show the Authoritativeness and the NED scores for the two solutions, on the given six Java software systems.

Looking at these results, the first observation is that the modified version of K-Medoids performs very well in terms of NED, better than any other considered clustering algorithm, with values always very close to 1. This is in some way an expected result, since the algorithm was designed with this goal.

As for the NED, we can notice that the introduction of the probabilistic model produces a very slight improvement in the quality of the proposed partitions for considered systems.

In this preliminary case study we did not identify any significant pattern between the term distribution across zones/weights and the results. For instance, the project with the highest results in terms of Authoritativeness is Jvlt, which is the one with fewer comments. Thus it seems that a careful wording for classes and methods could lead to more benefits than having many sparse comments. But on the other hand, also EasyMock and JabRef scored very well in terms of Authoritativeness, and both these systems present a
balanced number of extracted terms between comments and code. As a consequence, further analysis is needed, exploiting larger datasets, with projects coming from the industrial world. In any case, it is interesting to compare the results with and without the EM algorithm.

When compared with results presented by Bittencourt and Guerrero in [3], we can see that once more the results are better than any other technique, both in terms of Authoritativeness and NED, confirming the findings described in the previous section.

In particular, as for Authoritativeness, a remarkable improvement comes for the worst case, where the probabilistic approach obtains a score which is 25% better than the one obtained by the k-means. As for the best predictions, both K-Medoids and the probabilistic approach improve the best results reported in [3] by a 4%.

C. Threats to validity

To comprehend the strengths and limitations of our case study, the threats that could affect the validity of the observed results are presented and discussed as well. In our case the reliability of the used measures (i.e., Authoritativeness and non-extremity of cluster distribution) represents a critical issue that may affect the generalization of the obtained results. Also, the Authoritativeness results could be strongly affected by the used authoritative partition. The lower and upper limits to compute the NED measure represent a further threat to the validity of our study. To deeply investigate the effectiveness of the proposed approach we plan to assess the achieved results using precision and recall measures.

Another issue is represented by the implicit randomness of the clustering algorithm, whose first point is taken by chance. To reduce biases, we performed 30 runs for each system under investigation. Then we considered the mean NED and Authoritativeness values of these runs.

Another issue is represented by the software systems we have used in the experimental investigation. A further investigation on different and larger software systems is required. Note that also the fact of having used open source software systems could affect the results of the presented study. Accordingly, we plan to conduct a further investigation on different sized commercial software systems.

VI. CONCLUSIONS

The maintainability of legacy software systems is improved in case they have a well-documented architecture [35], but if this documentation becomes outdated, reverse engineering tools have to be employed to align it with the actual implemented software architecture. In this scenario, clustering based approach play an important role in the reverse engineering of software systems.

In this paper we have presented a novel clustering based approach to partition object oriented software systems implemented in Java into meaningful software subsystems. The approach, intended as an Eclipse plug-in, is based on a process that first extracts lexical information from the source code according to four zones: comments, Javadocs, class/method identifiers, and variable identifiers. The information provided by each zone is weighted using a probabilistic model and the Expectation-Maximization (EM) algorithm. As subsequent step, we clustered the data by a version of the well-known K-Medoids algorithm that we modified to better handle the specific domain.

In order to automate the approach and to facilitate its adoption, we have also implemented a prototype of a supporting system. The approach and the tool prototype have been validated in a case study constituted of six open source software systems implemented in Java. The results obtained on these software systems have been evaluated using two criteria: Authoritativeness and non-extremity cluster distribution. The data analysis has revealed that the combination of the modified K-Medoids, with the probabilistic approach scores very well in terms of NED and Authoritativeness. In particular, the introduction of the probabilistic model seems to bring particular benefits in reducing the range of the results. This is due to the fact that the worst results obtained with our proposal are sensibly better than the worst ones obtained with other clustering-based approaches.

As future work we also plan to extend the tool to analyze software systems implemented using different object oriented programming languages, e.g. C++ and C#. The possibility of considering information of the development process (e.g., analysis and design documentation) represents another possible direction to extend our work. Also, the possibility of using the original structure of the classes within the packages will be investigated.

Future work should be devoted to consider different criteria to deeply investigate the effectiveness and the correctness of the proposed approach. For example, the precision and recall measures could be employed to assess the overall quality of the identified partitions. To compute these measures experts are needed. Unfortunately, software engineers with a suitable experience on an open source software system are difficult to find and then to involve in empirical experimentation. Accordingly, we are going to conduct research collaborations with the software industries of our contact network to assess both the approach and the tool prototype on commercial software systems. This part of
ACKNOWLEDGMENT

The authors wish to thank Valerio Maggio and Ennio Masi for their support in the development of the solution proposed in this paper.

REFERENCES